MULTITASKING APPLICATION USING CRAY-2 ON ARNOLDI'S METHOD FOR COMPUTING A FEW EIGENVALUES IN A LARGE SPARSE MATRIX

San Kyung Kim and A.T. Chronopoulos

UMSI 88/142
December, 1988
Multitasking Application using CRAY-2 on Arnoldi’s Method for computing a few eigenvalues in a large sparse matrix

Sun kyung Kim
A. T. Chronopoulos

Computer Science Department
University of Minnesota
Minneapolis, MN 55455 USA

Abstract:
Multitasking using CRAY-2 is applied to Arnoldi method for computing a few eigenvalues and the corresponding eigenvectors in a large sparse matrix. Several numerical experiments are described, including comparisons of the results according to the number of processors and the size of matrices.

1. Introduction
Many important problems in engineering and science require the computation of a small number of eigenvalues with algebraically largest (or smallest) real parts of a large nonsymmetric real matrix A. The method of Arnoldi can successfully be used for solving eigenproblems of large nonsymmetric matrices. Like the symmetric Lanczos method, Arnoldi’s algorithm realizes a projection process onto the Krylov subspaces \( K_m(u_1, u_2, u_3, ..., u_m) \) spanned by \( u_1, Au_1, ..., A^{m-1}u_1 \), where \( u_1 \) is the initial vector. It reduces the given matrix A sequentially to an upper Hessenberg form. The approximate eigenvalues are obtained by computing the eigenvalues of the Hessenberg matrix \( H_m \) of order \( m \), produced at the \( m \)th step of Arnoldi’s process.
However, unlike the symmetric Lanczos algorithm, the growth of computational time and storage becomes excessive as the number of steps increases. Because of storage considerations, the dimension $m$ of the upper Hessenberg matrix $H_m$ cannot be chosen as large as necessary to ensure the desired accuracy. This difficulty may be overcome by using the incomplete orthogonalization process, but the basis is no longer orthogonal.

In sec.2 we describe the basic Arnoldi algorithm and the iterative Arnoldi algorithm and the deflated Arnoldi algorithm which has been used in the actual program.

2. The Method of Arnoldi

2.1 The basic Arnoldi algorithm

Let $v_i$ be a starting vector of norm one, and let $m$ be chosen not exceeding the dimension $N$ of the matrix $A$. A brief description of Arnoldi's method is the following:

Algorithm 2.1

For $j = 1, 2, \ldots, m$ do:

1. $w_{j+1} = Av_j - \sum_{i=1}^{j} h_{ij} v_i$ with $h_{ij} = (Av_j, v_j)_{i=1,2,\ldots,j}$

2. $h_{j+1,j} = ||w_{j+1}||$

3. $v_{j+1} = w_{j+1}/h_{j+1,j}$

This algorithm produces an orthogonal basis $V_m = \{v_1, v_2, \ldots, v_m\}$ of the Krylov subspace $K_m = \text{span}\{v_1, Av_1, \ldots, A^{m-1}v_1\}$. Let $V_m$ be the $N \times m$ matrix formed by the column vectors $v_1, v_2, \ldots, v_m$. Then the matrix $H_m = V_m^TAV_m$ is an upper $m \times m$ Hessenberg matrix with elements $h_{ij}$ given by algorithm(2.1). If the matrix $A$ is symmetric then $H_m$ reduces to a symmetric tridiagonal matrix and this algorithm reduces to the symmetric Lanczos method.
The eigenpair \( s^{(m)}, z^{(m)} \) satisfying

\[
z^{(m)} \in \text{span}[V_m]
\]

\[
(A-s^{(m)}I)z^{(m)} \parallel u_j, \quad j=1,2,...,m.
\]

are usually called Ritz values and Ritz vectors that are approximation to eigenvalues and eigenvectors. The Ritz values of \( A \) in \( K_m \) are the eigenvalues \( s_i^{(m)} \) of \( H_m \), and the Ritz vectors are the vectors \( V_m y_i^{(m)}(=z^{(m)}) \), where the \( y_i^{(m)} \) are the eigenvectors of \( H_m \) associated with the \( s_i^{(m)} \). The residual norms of the Ritz pair \( s^{(m)}, z^{(m)} \) can be computed by using the formula \( \| (A-s^{(m)}I)z^{(m)} \parallel = h_{m+1,m} | e_m^T y^{(m)} | \), where \( e_m \) is the \( m \)-dimensional vector \( e_m = (0,0,...,1)^T \). This is a direct consequence of the following equality, which derives from the algorithm(2.1):

\[
A V_m = V_m H_m + h_{m+1,m} v_{m+1} e_m^T
\]

Therefore the residual \( h_{m+1,m} v_{m+1} e_m^T \) can be used as an error term for checking accuracy in the actual program. It is therefore quite simple to check step by step whether the desired accuracy is attained and to stop as soon as it is.

2.2 Iterative Arnoldi Method

On the practical side, there remains the problem of choosing the number \( m \) of steps. The storage of the Hessenberg matrix requires about \( \frac{1}{2} m^2 \) locations, and the storage of orthogonal basis \( V_m \) requires \( N \times m \) locations. When \( N \) is large, this may become impossible if we consider that convergence is often achieved for values of \( m \) such as \( m \approx \sqrt{N} \). Therefore the number of steps, \( m \), is limited by the available main memory. After computing the Ritz values and Ritz vectors with the maximum possible \( m \) according to main memory capacity, one may find the eigenelements have not converged to the desired accuracy. The simplest way to overcome this difficulty is to repeat the process with \( v_i \) replaced by an Ritz vector or a combination of Ritz vectors.
The iterative algorithm for computing 1 dominant eigenvalues is the following:

**Algorithm 2.2**

1. Choose the number of Arnoldi steps \( m \) and initial vector \( v_1 \)
2. Construct \( V_m \) and \( H_m \) by algorithm 2.1, and compute the dominating Ritz value \( s_1^{(m)} \) and the associated Ritz vector \( z_1^{(m)} \)
3. If the pair \( s_1^{(m)}, z_1^{(m)} \) is sufficiently accurate, then stop. Otherwise take \( v_1 = z_1^{(m)} \) and go back to 2.

For simplicity the above algorithm is described only for the problem of computing one eigenvalue and the associated eigenvector. \( P \) dominant eigenvalues can be computed by the iterative Arnoldi algorithm as well. One should then take as a new starting vector \( v_1' \) in step 3 a linear combination of \( z_1^{(m)}, z_2^{(m)}, \ldots, z_P^{(m)} \).

**2.3 Deflated Arnoldi Method**

The starting method may encounter some difficulties especially in cases when the number of eigenvalues to be computed is not small. The restarting process is often unable to keep the accuracy gained in the previous steps for all eigenvalues. i.e. the accuracy may improve in some eigenvalues but deteriorates in some others. It is difficult when the number of eigenvalues is not small to make the method produce a similar accuracy for all eigenvalues. This is why deflation is so important.

Saad[2] describes a deflation technique which is a simple variation of Wielant's deflation and shows it is very suitable for computing orthonormal bases of invariant subspaces and corresponding partial Schur forms. A partial Schur factorization is of the form

\[ AQ = QR \]
where $Q$ is an $N \times p$ complex unitary matrix and $R$ is upper triangular matrix.

Let $u_1, u_2, \ldots, u_p$ be a set of Schur vectors associated with the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$. We denote by $U_p$ the matrix of columns vectors $u_1, u_2, \ldots, u_p$. $U_p$ is an orthonormal matrix whose columns form a basis of the eigenspace associated with eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$. Let $\Sigma_p$ be the $p \times p$ diagonal matrix $\Sigma_p = \text{Diag}\{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ where $\sigma_i, i = 1, \ldots, p$ is chosen so that the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_{i-1}$ will in return be the ones with largest real parts during the algorithm. Then the eigenvalues of the matrix

$$A_p = A - U_p \Sigma_p U_p^H$$

are $\lambda_i' = \lambda_i - \sigma_i$ for $i < p$ and $\lambda_i'' = \lambda_i$ for $i > p$. Let $A U = U R$ be the Schur factorization of $A$. We have $A_p U = U (R - E \Sigma_p E_p^H)$ where $E = \{e_1, e_2, \ldots, e_p\}$. This suggests a very simple incremental deflation procedure consisting of building the matrix $U_p$ one column at a time. Thus at the $p$-th step, once the eigenvector $\phi_{p+1}$ is computed by the iterative Arnoldi algorithm we can orthonormalize it against all previous $u_i$'s to get the next Schur vector $u_{p+1}$ which will be appended to $U_p$ to form the new deflation matrix $U_{p+1}$. Clearly, $u_{p+1}$ is a Schur vector associated with the eigenvalue $\lambda_{p+1}$. The deflated Arnoldi method can be used in conjunction with the iterative Arnoldi algorithm (2.1). The deflated algorithm for computing $P$ dominant eigenvalues is the following:

Algorithm 2.3

1. Choose the number of Arnoldi steps $m$ and initial vector $u_1$

2. Construct $V_m$ and $H_m$ by algorithm 2.1, and compute the dominating Ritz values $s_1^{(m)}, s_2^{(m)}, \ldots, s_p^{(m)}$ and associated Ritz vectors $z_1^{(m)}, z_2^{(m)}, \ldots, z_p^{(m)}$

3. When the pairs $s_i^{(m)}, \ldots, s_l^{(m)}$, $l \leq p$, are sufficiently accurate, if $l = p$ then stop otherwise $u_{l+1} = z_{l+1}^{(m)}$ and go back to 2.
3. Arnoldi method applications on CRAY-2

3.1 Vectorization

The CFT77 compiler produces CRAY-2 machine language instructions from Fortran language statements is particularly effective in compiling statements describing vector processing. When properly applied, vector processing allows dramatic decreases in computation time over equivalent scalar processing method. Not all DO loops are vectorizable. For example, the following loops are part of Saad's code implementing the deflated Arnoldi method.

\[
\begin{align*}
\text{DO } 1 & \ J=1,N \\
\text{DO } 2 & \ II=1,NEV1 \\
T &= 0.0 \\
K1 &= \text{IORD}(II) \\
\text{DO } 3 & \ K=1,IM \\
T &= T + U(J,K) \ast \text{EVEC}(K,K1) \\
3 & \text{ CONTINUE} \\
2 & \ V(II) = T \\
\text{DO } 4 & \ II=1,NEV1 \\
4 & \ U(J,\text{ILOCK}+II-1) = V(II) \\
1 & \text{ CONTINUE}
\end{align*}
\]

The nested DO loops above are not suitable good for vectorization, this routine can be modified as following:

\[
\begin{align*}
\text{DO } 1 & \ II=1,NEV1 \\
K1 &= \text{IORD}(II) \\
\text{DO } 2 & \ K=1,IM \\
\text{DO } 3 & \ J=1,N \\
3 & \ UTEMP(J,II) = UTEMP(J,II) + U(J,K) \ast \text{EVEC}(K,K1)
\end{align*}
\]
We can obtain considerable speedup in vectorizable DO loops after the above mentioned modification. The speedup of the nested loop code before the modification versus the modified vectorizable code is shown in Fig. 1. Also we obtain considerable speedup after we modify Saad's original program which consists of the iterative and deflated Arnoldi method. The speedup of the original code versus the modified one is shown in Fig.2.

3.2 Multitasking application and Numerical experiments

The code that is used is Saad's implementation of deflated Arnoldi method. Actually, these multitasked codes are used in conjunction with the Saad's program which is modified for vectorization. Most part of Arnoldi algorithm consists of dot product \((x,y), \text{SAXPY } y = t*x + y\) and matrix by vector multiplication. So, it is supposed to get the speed improvement in multitasked Arnoldi method when we get good speedup in multitasked dot product, \text{SAXPY}, matrix by vector multiplication. We used 2.4 processors for multitasking to compare with the case of single processor on CRAY-2.(Fig. 4,5,6) In matrix by vector multiplication, the Laplace matrix[6] is used. The structure of \(n \times n\) Laplace matrix is the same as that of tridiagonal matrix with each element replaced by a \(m \times m\) matrix block as following(\(n = m^n\)):

\[
A = [C_{k-1}T_kC_k], \quad 1 \leq k \leq m
\]

where \(T_k, C_k\) are matrices of order \(m\); and \(C_0 = C_m = 0\). \(A\) is weakly diagonally dominant. The blocks have the form
\[ C_i = diag[c_1^k, \ldots, c_m^k] \]

\[ T_i = [b_i^{-k}, a_i^k, b_i^k], \quad 1 \leq i \leq m, \]

with \( b_i^k < 0, \quad c_i^k < 0, \quad b_0^k = b_m^k = 0, \) and \( a_i^k > 0. \)

The speedup is best in case of matrix by vector multiplication because this has the largest number of operations, and when dense matrix is applied we can get better speedup. The speedup in dot product is a little bit poorer than those in the other cases.

As an application matrix for Arnoldi method, Laplace matrix is used and we can get speedup similar to Fig 1,2,3. We have set the parameter indicating the number of wanted eigenvalue to \( \text{NEV}=2 \), the number of Arnoldi steps to \( \text{LARN}=10 \), the stopping criterion to \( \text{EPS}=10^{-4} \), the starting vector with norm 1. (Fig. 7)

The slow convergence in Arnoldi type methods, comes from the poor separation of the desired eigenvalues of some matrix.[2] We compared the matrix by vector multiplications according to residual norms with same matrix size \( 32^2 \) and single processor. Here, three kind of matrices are used. first, Laplace matrix is used (matrix A in Fig. 8). Second, tridiagonal matrix with 2 in diagonal and -1 in subdiagonal is used(matrix B in Fig. 8). Third, the transition matrix[3] of the Markov chains describing a random walk on an \( (n+1) \times (n+1) \) triangular grid(matrix C in fig. 8). A transition may take place from the node \( (i,j) \) to one of the four adjacent nodes \( (i+1,j), (i,j+1), (i-1,j), (i,j-1) \). The probability of jumping from the node \( (i,j) \) to either of the nodes \( (i+1,j) \) or \( (i+1,j) \) is

\[ p_{u(i,j)} = 0.5 - \frac{(i+j)}{2n} \]

(These transition can occur only for \( i+1 \leq n \) and \( j+1 \leq n \).) The probability of jumping from the node \( (i,j) \) to either of the nodes \( (i,j-1) \) or \( (i-1,j) \) is
\[ pd(i,j) = \frac{(i+j)}{2n} \]

this probability being doubled if either \(i\) or \(j\) is 0. The nodes are numbered in the order: \((0,0), (1,0), \ldots, (n,0), (0,1), (1,1), \ldots\). The detailed description of the third matrix is in [3].

2. Summary

We have multitasked a deflated Arnoldi code [Saad 1,2], and run tests on the four processor CRAY-2 of the University of Minnesota Supercomputer Institute. We used multitasked subroutines for the three vector operations used in the program: dot product, vector update and matrix vector product. Our tests show that the overall speedup of the program is more than two for large dimensions. The speedup for the matrix vector product and the vector update is more than three. The dot product and some serial code fraction seem to limit the overall performance of the Arnoldi code. Local memory management would probably increase the performance of the code.
Fig. 1.: CRAY-2(SC) speedup on vectorization of DO loop in sec. 3.1
Fig. 2.: CRAY-2(SC) performance on Arnoldi method with vectorization
Fig. 3. : CRAY-2(SC) performance on dot product, SAXPY and matrix-vector multiplication.
Fig. 4.: CRAY-2(SC) speedup on dot product
Fig. 5.: CRAY-2(SC) speedup on SAXPY
Fig. 6.: CRAY-2(SC) speedup on matrix by vector multiplication
Fig. 7.: CRAY-2(SC) speedup on Arnoldi method
Fig. 8.: comparisons of matrix-vector multiplications on Arnoldi method
References.


program tdarn

c test program for dar0.f

c the matrix is Laplace matrix.
c mgrid in the common ope0 represents the block size of matrix

c parameter (n=12**2)
dimension u(n,1l2),ul2(n),ul1(n),v(n),ue(1048576,111)
dimension a(-1023:1049600),b(-1023:1049600),c(-1023:1049600)
dimension rr(30,30), evcc(30,30), wr(30), wi(30), res(30)
integer iord(30),ird(30)
common /ope0/ icount,mgrid
common /ope1/a,b,c
common /opevc/ u

data iarn/10/,eps/1.e-06/, k/12/, iout/6/, maxope/1000000/
data nev/2/

c nrr = 30

c initialize common ope0 for matrix vector operations.
mgrid = k
icount = 0

c size of problem
c n = k+2

110 print heading
write(iout,101) k, n
c fill first vector of u(1,*) with random vector ... il = 19241
do 12 k=1,n
12 u(k,1) = ranf()
c initialize matrix A
do 113 l=1,n
a(i,j) = 4.0
b(i,j) = 1.0
c(i,j) = 1.0
112 continue
do 113 l=1,mgrid
br(i,m) = 0.0
c(n-l+1) = 0.0
113 continue

c call subroutine arnchb

top=0
tmpa=0
tiel=0
tddo=0
tdxap=0
tl=irts()**4.2e-09

C------------------------ NOTES ------------------------
C H E S S E N B A U R G ARNOLDI MATRIX. EVEC = EIGENVECTORS OF HH ...
C I M = FIRST DIMENSION OF HH, EVEC.
C U S IS OF DIMENSION N*(IM+1) AT LEAST ...
C------------------------------ FIRST EXECUTABLE INSTRUCTION --------------------------

IM = IARN+1
SIG = REAL(ISIG)
IV = 1 + N*(IARN+1)
IH = IV+N
IEVEC = IMH + IH*IARN
IMH=IMH*IARN+1

C-----------------------------------------------
ITS = 0
ILOCK = 0
CALL ZERMAT(NRR, NRR, RR)
10 ILOC1 = ILOCK+1
C-- NORMALIZE U(*,1) -------------------------------
CALL MGVSR(N, ILOC1, U12, tddo, tdxap)

C---
C CALL ZERMAT(IM, IH, U13)
C CALL COPMAT (ILOCK, RR, NRR, U13, IH)

C-----------------------------------------------
111 = ILOC1
C------------------------ AR N O L D I L O O P ------------------------
58 I = I1
I1 = I + 1
K = 1+N*(I-1)
tope1 = irts()**4.2e-09
CALL OPE (N, U1,I), U(1,I+1)
tope2 = irts()**4.2e-09
tope=tope2-tope1
top=top+tope
ITS = ITS + 1
K = IMH+IH*(I-1)
tmpai = irts()**4.2e-09
CALL MGVSR (N, I1, U13(K-IMH+1), tddo, tdxap)
kk=K-IMH

data iarn/10/
MODIFIED GRAM - SCHMIDT WITH PARTIAL REORTHO. THE VECTOR SS(*,I1) IS
ORTHOGONALIZED AGAINST THE FIRST I VECTORS OF SS (WHICH ARE ALREADY
ORTHOGONAL). THE COEFFICIENTS OF THE ORTHOGONALIZATION ARE RETURNED IN
THE ARRAY R

TET=10.0e0
DO 53 J=1, II
53 U13(KK+J) = 0.0e0
I2 = I1+1
IT = 0

HINORM = 0.0e0
IT = IT +1
IF (IT.EQ. 0) GOTO 56
DO 55 J=1, I2
tddotl=irtc(1)**4.2e-09
call dott(u.u,l1),u(1,ii),summ)
tddot2=irtc(1)**4.2e-09
tddot=tddot2*tddot1
tddot=tddot+tddot
TT=summ
HINORM = HINORM + TT*2
U13(KK+J) = U13(KK+J) + TT
tdxpl=irtc(1)**4.2e-09
call axpy(n,-TT,u(1,1),u(1,ii))
call daxpy(n,-TT,u(1,1),1.,u(1,ii),1.,1.)
tdxpl=dxpl+tdxpl
tdxaxl=tdxaxl+tdxax
55 CONTINUE
56 call dott(u.u,l1),u(1,ii),summ)

TT=summ
c TEST FOR REORTHOGNALIZATION SEE DANIEL ET. AL.
c TWO REORTHOGNALIZED ALLOWED ---

IF (TT.TET .LE. HINORM .AND. IT .LT. 2) GOTO 54
TT = sqrt(TT)
U13(KK+I) = TT
IF (IT .EQ. 0.0e0) goto 111
TT = 1.0e0/TT
DO 57 JK=1,N
57 U(jk,II) = U(jk,II)*TT

/ 
111 
tmgsar2=irtc(1)**4.2e-09
tmgs=tmgsar*tmgsar

K = 1+IARN - IJVEC + IH*(J-1) + J-1
U13(K-IHH+I) = 1.0e0

CALL ZERNAT(IH, IH, U13(IHI))
DO 2 J=1,IARN

C END OF ARNOLDI LOOP. NOW PREPARE TO CALL QR.----------------------
CALL ZERNAT(IH, IH, U13(IHI))
DO 2 J=1,IARN

2

CALL ORDS (IARN, WR, WI, IORD, U12)

C----------------------- PRINT OUT RESULTS --------------------------
CALL PRES(NEV, WR, WI, U12, IOUT)
ICONV = 0
DO 81 J = ILOC1, IARN
IF (U12(J) .LE. EPS) ICONV = ICONV + 1
81 CONTINUE

C -- GET APPROXIMATE CENTER OF CONVEX HULL ---------------------
DC = 0.0e0
DO 79 J=ILOC1, IARN
DC = DC + WR(J)
DO 79 J=ILOC1-1, IARN
C----- SAVE RESIDUAL RESIDUAL NORMS -----------------------------
DO 791 J=1, ICONV
res(ILOCK+J) = U12(ILOCK+J)
791 CONTINUE

C GET THE LINEAR COMBINATION OF EIGENVECTORS TO BE USED IN NEXT
ITERATION ... X = NEXT REAL(EIGENVECTORS) ABOUT TO CONVERGE.

J0 = ILOCK + ICONV + 1
K0 = IORD(J0)
DO 83 III=1, IARN
T = 0.0e0
J = J0
III = 0
82 IF (WI(J) .LT. 0.0e0 .OR. (WR(J)-DC)*SIG .LT. 0.0e0) GOTO 825
K = IVECE + IH*(IORD(J)-1)+III-1
T = T+U13(K-IHH+I)*U12(J)
J = J1 + 1
825 J = J1
IF (J1 .LT. 1) GOTO 82
83 END
K = IVEC + I-H*(K1+1-I)*I+1
U3(JH-IH)+1 = T
J0 = ICONF+1

C teivelirtc() *4.2e-09
C CALL EVECT (N, J0, IARN, ILOC1, U, U13(IHI), IH, IORD(ILOC1), U12)
K = IHI
DO 193 J = 1, IH
DO 193 K = 1, IH
EVEC(J, II) = U13(KK)
K = KK+1
193 CONTINUE
DO 114 II = ILOC1, J0
114 IORD(II-ILOC1+1) = IORD(II)

C-------------------------COMPUTES RITZ VECTORS FOR THE ARNOLDI METHOD-----
C COMPUTES EIGENVECTORS 1.2.---- NEV1 AND PUTS THEM STARTING IN
C COLUMN NUMBER ILOC1 OF ARRAY U(N,*)
C
C teivelirtc() *4.2e-09
C IF (J0.LE.0) GOTO 192
DO 351 II = 1, J0
KL = IORD(II)
DO 351 K = 1, IARN
DO 351 J = 1, n
U = U(J, II) + U(J, K)*EVEC(K, KL)
351 CONTINUE
TT = EVEC(K, KL)
CALL ASYMP(N, TT, U(1, K), U(1, II))
351 CONTINUE
DO 352 II = 1, J0
KKK = ILOC1+II-1
DO 352 J = 1, n
U(J, KKK) = U(J, II)
352 CONTINUE
K = IHI
DO 115 J = 1, II
DO 115 K = 1, II
115 U13(KK) = EVEC(II, JJ)
K = KK+1
115 CONTINUE
DO 116 II = ILOC1, J0
116 IORD(II) = IORD(II-ILOC1+1)

C UPDATE A FEW THINGS IF SOME EIGENVALUES HAVE CONVERGED...
C IF (ICNY .LE. 0) GOTO 11
DO 84 J = 1, ICONF
K = ILOC1+J-1
CALL HSQR (N, X, U12, tddo, tdaxp)
84 CONTINUE

C CHECKING THE ACTUAL RESIDUAL NORMS --
C
C teivelirtc() *4.2e-09
K = ILOC (ILOCK + 1)
IF (ILOCK .GT. NRR) GOTO 85
C---- STORE INFORMATION ON EIGENVALUES AND RESIDUAL VECTORS -------
K = N*(ILOCK-1) + 1
CALL OPE(N, U1L, ILOC1, U12)
JJ = ILOCK
IF (WT(JJ) .GT. 0.0e0) JJ = JJ+1
DO 9 I = 1, JJ
K = N*(I-1)+1
CALL OPE(N, U1L, U1L, SUMM)
RR(I, ILOCK) = SUMM
9 CONTINUE
C((((((((( IF CONVERGED THEN LEAVE )))))))))))))))
11 IF (IT2 .LT. MAXOPE AND. ILOCK .LT. NEV) GOTO 10
NEV = ILOCK

C
C---------------------------------------------------
C 999 WRITE (*,108) (WR(j), WI(j), RES(j), J=1,NEV)
C WRITE (*, 201) IECOUNT
C
C now we compute eigenvectors by calling ortgal.
C WRITE (*,*)' Computing eigenvectors'
C tt=irtc() *4.2e-09
C CALL ORTGA(N, NEV, NEV, U, V, WR, WI, RR, EVEC, NRR, IORD)
C checking the actual residual norms --
C
C 111 CALL ACRES(N, NEV, U, V, RES, WR, WI)
END OF MAIN ROUTINE

subroutine ope(n,v,av)
  real a(-1023:1049600), b(-1023:1049600), c(-1023:1049600)
  real v(1048576), av(1048576)

  c performs y = a x for

  common /ope0/ icount, m
  common /ope/ a, b, c

  c dimension n of matrix is (m**2)
  c icount: used to count the number of operations y=ax...

  c multitasking
  external ax, axl, axn
  integer task1(2), task2(2), task3(2), task4(2)
  common/tcas/task1, task2, task3, task4

  c - n must be devisable by 4
  c - nlength=n/4
  istart1=1
  istart2=istart1+nlength
  istart3=istart2+nlength
  istart4=istart3+nlength
  lend1=nlength
  lend2=lend1+nlength
  lend3=lend2+nlength
  lend4=lend3+nlength

  c - initialize task array -
  task1(1)=2
  task2(1)=2
  task3(1)=2
  task4(1)=2

  c - start 4 tasks - each to form 1/4 of the dot product x.y
  call tskstart(task1, ax, istart1, lend1, m, v, av)
  call tskstart(task2, ax, istart2, lend2, m, v, av)
  call tskstart(task3, ax, istart3, lend3, m, v, av)
  call tskstart(task4, ax, istart4, lend4, m, v, av)

  c - wait for tasks to complete -
  call tskwait(task1)
  call tskwait(task2)
  call tskwait(task3)
  call tskwait(task4)

  c icount=icount+1

  return
end

subroutine ax(istart, lend, m, v, av)
  real a(-1023:1049600), b(-1023:1049600), c(-1023:1049600)
  real v(-1023:1049600), av(1048576)
  common /ope/ a, b, c

  do i=istart, lend
    av(i)=a(i)*v(i)+b(i)*v(i+1)+c(i)*v(i+1)
    / + c(i)*v(i+m)+c(i-m)*v(i-m)
  end
1 continue

c  return
  end

c-----------------------------------------END OF SUBROUTINE OPE-----------------------------------------
c

SUBROUTINE COPMAT (JCOL, A, NA, B, NB)
c COPIES J COLUMNS OF MATRIX INTO SAME COLUMNS OF B
real A(NA,1), B(NB,1)
  IF (JCOL .LE. 0) RETURN
  DO 2 J=1, JCOL
       DO 3 I=1, NA
          B(I,J) = A(I,J)
 3 RETURN
  END

SUBROUTINE ZERMAT (N1,N2,A)
C FILLS MATRIX A WITH ZEREOES
DIMENSION A(1)
  NN = N1*N2
  DO 1 I=1,NN
     A(I) = 0.0e0
 1 RETURN
END

subroutine saxpy(n,t,x,y)
  real x(1), y(1), t
C DOES THE FOLLOWING OPERATION
  y (+- t * x)
c multitasking
  external saxp
  integer task1(2),task2(2),task3(2),task4(2)
  common/tcas/task1,task2,task3,task4
  common/start/nlength

  n must be devisable by 4 -
  nlength=n/4
  istart1=1
  istart2=istart1+nlength
  istart3=istart2+nlength
  istart4=istart3+nlength
  - initialize task arrays -
     task1(1)=2
     task2(1)=2
     task3(1)=2
     task4(1)=2

  - start 4 tasks - each to form 1/4 of the dot product x.y
     call tskstart(task1,saxp,istart1,x,y,t,mhalf)
     call tskstart(task2,saxp,istart2,x,y,t,mhalf)
     call tskstart(task3,saxp,istart3,x,y,t,mhalf)
     call tskstart(task4,saxp,istart4,x,y,t,mhalf)
  - wait for tasks to complete -
     call tskwait(task1)
     call tskwait(task2)
     call tskwait(task3)
     call tskwait(task4)
  c
  return
  end

subroutine saxp(istart,x,y,t)
  real x(1),y(1)
  common/start/nlength
  do 1 i=istart,istart+nlength-1
    y(1)=y(1)+t*x(i)
 1 return
  end

END OF SUBROUTINE AXPY-----------------------------------------

COMPUTES THE INNER PRODUCT T=(X,Y)
  subroutine dott(a,x,y,summul)
real x(1), y(1)

multitasking
external dotprod
integer task1(2), task2(2), task3(2), task4(2)
common/tcas/task1, task2, task3, task4
common/start/lnlength
common/dotp/sumxy(4)

n must be devisible by 4
length=n/4
istart1=1
istart2=istart1+length
istart3=istart2+length
istart4=istart3+length

initialize task arrays

task1(1)=2
task2(1)=2
task3(1)=2
task4(1)=2

start 4 tasks - each to form 1/4 of the dot product x.y
call tskstart(task1, dotprod, istart1, l.x.y, l.length)
call tskstart(task2, dotprod, istart2, 2.x.y, l.length)
call tskstart(task3, dotprod, istart3, 3.x.y, l.length)
call tskstart(task4, dotprod, istart4, 4.x.y, l.length)

wait for tasks to complete -
call tskwait(task1)
call tskwait(task2)
call tskwait(task3)
call tskwait(task4)

summul=0
100 do 1 i=1, 4
100 summul=summul+sumxy(i)
return
end

-----------------------------------------------

subroutine dotprod(istart, no, x, y)

real x(1), y(1)
common/start/lnlength
common/dotp/sumxy(4)
sumxy(no)=0
1 do i=istart, istart+length-1
2 sumxy(no)=sumxy(no)+x(i)*y(i)
return
end

-----------------------------------------------

SUBROUTINE MGSR (N, II, SS, R, tddo, tdxp)

MODIFIED WITH PARTIAL REORTH. THE VECTOR SS(*, II) IS
ORTHOGONALIZED AGAINST THE FIRST I VECTORS OF SS (WHICH ARE ALREADY
ORTHOGONAL). THE COEFFICIENTS OF THE ORTHOGONALIZATION ARE RETURNED IN
THE ARRAY R

real SS(N, I), R(I), HINORM, TET, T, sqrt
DATA TET/10.0e0/
DO 53 J=1, II
53 R(J) = 0.0e0
   T = TET
   IT = 0
54 HINORM = 0.0e0
   IT = IT +1
   IF (I .EQ. 0) GOTO 56
   DO 55 J=1, I
   55 tdott1=irtc()*.4e-09
   call dott(n, ss(l, j), ss(1, II), summ)
   tdott2=irtc()*.4e-09
   tdott=tdott1-tdott2
   tddo=tddo+tdott
   T = summ
   HINORM = HINORM + T**2
   R(J) = R(J) + T
   taxpy1=irtc()*.4e-09
   call axpy(n,-.5, ss(l, j), ss(1, II))
   taxpy2=irtc()*.4e-09
   taxpy=taxpy2-taxpy1
   tdxp=tdxp+taxpy
   CALL SAXPY(N,-.5, ss(l, j), ss(1, II), 1)
   CONTINUE
   56 call dott(n, ss(l, II), ss(1, II), summ)
   T = summ
C TEST FOR REORTHOGONALIZATION SEE DANIEL ET. AL.
C TWO REORTHOGONALIZATION ALLOWED ----
IF (T**T .LT. HINORM AND. IT .LT. 2) GOTO 54
T =sqrt(t)
R(II)= T
IF (T .EQ. 0.0e0) RETURN
T = 1.0e0/T
DO 57 K=1, N
57 SS(K, II) = SS(K, II)*T
RETURN
END

SUBROUTINE ORDRS (N, ER, EI, IORD, RES)

* PERMUTES THE E.V.'s IN SMALLEST RESIDUAL NORMS.
* INPUT ER = REAL PARTS OF EIGENVALUES
*     EI = IMAG. PARTS OF EIGENVALUES
*     RES = RESIDUAL NORMS ASSOCIATED WITH THE EIGENVALUES (WR, WI)
* OUTPUT: IORD IS THE PERMUTATION. Y CONTAINS ORDERED N Ri.
*         AND ER, EI, RES ARE ORDERED CORRECTLY.

DIMENSION ER(1), EI(1), IORD(1), Y(1), RES(1)

GET PERMUTATION FIRST.
IF (N.LE. 1) RETURN
RMAX = sqrt(ADOT(N, RES, 1, RES, 1))
DO J = 1, N
   XM = RMAX +1.0e0
   DO J = I, N
      X = RES(J)
      IF (X.GE.XM) GOTO 2
      XM = X
      I = J
   CONTINUE
   K = IORD(JM)
   IORD(JM) = IORD(I)
   IORD(I) = K
   T = ER(IM)
   ER(IM) = ER(I)
   ER(I) = T
   T = EI(IM)
   EI(IM) = EI(I)
   EI(I) = T
   T = RES(IM)
   RES(IM) = RES(I)
   RES(I) = T
CONTINUE
C SWAP DISORDERED EIGENVALUES ...
CALL SWAPS (N, ER, ET, RES, IORD)
RETURN
END

SUBROUTINE SWAPS (N, ER, EI, RES, IORD)
REAL ER(1), EI(1), RES(1), XSAV
INTEGER IORD(1), I, J, N

SWAPS DISORDERED EIGENVALUES SO THAT...
IN A COMPLEX PAIR THE FIRST ONE APPEARS...
WITH A POSITIVE REAL PART...

I = 0
I = I + 1
IF (EI(I) .EQ. 0.0e0) GOTO 6
IF (EI(I) .GT. 0.0e0) GOTO 5
BAD ORDER ----> SWAP
XSAV = EI(I)
EI(I) = EI(I+1)
EI(I+1) = XSAV
XSAV = RES(I)
RES(I) = RES(I+1)
RES(I+1) = XSAV
J = IORD(I)
IORD(I) = IORD(I+1)
IORD(I+1) = J
I = I + 1
6 IF (I .LT. N) GOTO 4
END

SUBROUTINE EVECT (N, NEVL, IM, ILOCK, U, EVEC, IN, IORD, V)
DIMENSION U(N,1), EVEC(IH, IM), IORD(1), V(1), UE(1048576,11)
COMMON /PEVCE/ UE

COMPUTES RITZ VECTORS FOR THE ARNOLDI METHOD.
COMPUTES EIGENVECTORS 1.2, NEVL AND PUTS THEM STARTING IN
COLUMN NUMBER ILOCK OF ARRAY U(N,?)
do 351 k=1,im
   tt=evc(k,kl)
   call axpy(n,tt,u(k),ue(1,ii))
351 continue
   
   c do 353 ii=1,nevl
     kkl=lock+ii-1
   do 354 j=1,n
     u(j,kkl)=ue(j,ii)
     ue(j,ii)=0.0
353 continue
   return
end

SUBROUTINE EIVECT (M, EVEC, IM, BETA, RES, WI)
C
C COMPUTES THE RESIDUAL NORMS OBTAINED FOR RITZ VECTORS IN AN ARNOLDI
C PROCESS. USES EIGENVECTORS OF THE HESSIAN MATRIX OF SIZE M.

J = 1
420 IF (WI(J) .EQ. 0.0e0) GOTO 419
   T = sqrt(sdot(M,EVEC(1,J),L,EVEC(1,J),1))
   IF (WI(J) .LE. 0.0) GOTO 421
   RES(J) = BETA*sqrt((EVEC(M,J)**2+2*EVEC(M,J)**2)/T)
   RES(JL) = RES(J)
   J = JL
   GOTO 421
C-------------------------------- 0 COMPLEX CASE  0-----------------
   J = J + 1
   T = T + sdot(M, EVEC(1,1),L,EVEC(1,1),1)
   RES(J) = BETA*sqrt((EVEC(M,J)**2+2*EVEC(M,J)**2)/T)
   RES(JL) = RES(J)
   J = JL
   GOTO 421
C-------------------------------- 0 REAL CASE  0-----------------
421 J = J + 1
   IF (J .LE. M) GOTO 420
   RETURN
end

C-------------------------------- 0 END OF RESIDL 0-----------------

C
C SUBROUTINE PRRES (NEVL, WR, WI, RES, IOUT)
C REAL WR(1), WI(1), RES(1)
C IF (IOUT .LE. 0) RETURN
C WRITE (IOUT,996)
C WRITE (997,997)
C DO 106 J=1,NEVL
C WRITE (IOUT,999) WR(J), WI(J), RES(J)
C 999 FORMAT(1H,2D23.10,1H,2D23.10,1H,2D23.10,1H,2D23.10,1H,2D23.10,1H,2D23.10)
C 106 CONTINUE
C WRITE (IOUT,996)
C 996 FORMAT(1H,7X,1H,6L1H)
C RETURN
end

C--------------------------------  END OF PRRES-----------------

C
C SUBROUTINE ORTGET (N, M, NEV, U, V, WR, WI, HH, EVEC, IH, IORD)
C
C THIS PROCEDURE COMPUTES THE SET OF EIGENVECTORS FROM SCHUR VECTORS
C THE SCHUR EIGENVECTORS AND NORMAL VECTORS SPANNING THE INVARIANT
C SUBSPACE ARE STORED IN U(*,J), J=1,M. THE PROCEDURE COMPUTES ONLY
C THE NEV FIRST EIGENVECTORS.
C N = SIZE OF THE PROBLEM
C M = DIMENSION OF THE INVARIANT SUBSPACE = NUMBER OF VECTORS IN U
C NEV = NUMBER OF DESIRED EIGENVECTORS
C U = SCHUR VECTORS ON INPUT. OUTPUT: FIRST NEV COLUMNS CONTAIN
C THE EIGENVECTORS.
C WR, WI (OUTPUTS) = EIGENVALUES (REAL PARTS, IMAGINAR PARTS)
C HH = IH X IH MATRIX CONTAINING THE SCHUR MATRIX. (IH >= M)
C IH = IH X IH WORK MATRIX WHERE IN MUST BE >= M
C IORD = INTEGER WORK ARRAY OF LENGTH AT LEAST M
C NOTE: THE ORDER OF THE EIGENVALUE/EIGENVECTORS MIGHT BE CHANGED.
C (TO BE FIXED)
C
C real U(N,1), V(M,1), HH(IH,1), EVEC(IH,1), WR(1), WI(1)
C integer IORD(1)
C
C CALL ZERMAT (IH, 1, EVEC)
C DO 3 J =1, M
C   IORD(I) = J
C   EVEC(I,J) = 1.0e0
C 3 CONTINUE
C CALL HQR
C CALL HQR2 (IH,N,1,HH,WR,WI,EVEC,EIVCT)
C CALL HQR3 (IH,N,1,HH,WR,VI,EIVCT)
C CALL GET EIGENVECTORS
C tets1=lrtc()*.2e-09
C CALL EIVECT(N,NEV,1,HH,EVEC,IH,IORD,V)
C tets2=lrtc()*.2e-09
Page 8
SUBROUTINE ACTRES (N, NEV, U, V, RES, WR, WI)
C----------------------------------------------------------
C------ OUTPUTS ACTUAL RESIDUAL NORMS PRODUCED BY PROJECTION
C------ METHODS
C------ USES ONLY ONE EXTRA WORK VECTOR V(N) NEEDED
C------ OUTPUTS ASSOCIATED RESIDUAL NORMS (ESTIMATED ONLY) IN RES(*)
C------
C------ real U(N,1), V(N), WR(1), WI(1), RES(1), XMU, TETA, T
C----------------------------------------------------------

J = 1

XMU = WR(J)
ttopel = irtc() * 4.3e-09
CALL OPE (N, U(1,J), V)
ttopel = ttopel - ttopel
print *, 'opel=', ttopel
CALL axpy (N, XMU, U(1,J), V)
DO 82 II = 1, N

V(II) = V(II) - XMU * U(II,J)
C---CASE OF A REAL EIGENVALUE.
IF (WR(J) .NE. 0.0e0) GOTO 820
ttdot1 = irtc() * 4.2e-09
call dott(N, v, v, sum1)
call dott(N, u(1,J), u(1,J), sum2)
T = sum1 / sum2
ttdot2 = irtc() * 4.2e-09
ttdot = ttdot2 - ttdot1
print *, 'tdot2=', ttdot
RES(J) = sqrt(T)
GOTO 829
C---CASE OF A COMPLEX EIGENVALUE.
J1 = J + 1
TETA = WI(J)
call axpy(N, TETA, U(1,J), V)
DO 821 K = 1, N

V(K) = V(K) + TETA * U(K, J1)
call dott(N, V, V, summa)
T = summa
CALL OPE (N, U(1, J1), V)
call axpy(N, XMU, U(1,J1), V)
call axpy(N, TETA, U(1,J1), V)
DO 822 K = 1, N
call dott(N, V, V, summa)
T = T + summa
call dott(N, U(1,J1), U(1,J1), sum1)
call dott(N, U(1,J), U(1,J), sum2)
T = T / (sum1 + sum2)
T = sqrt(T)
RES(J) = T
RES(J1) = T
J = J1

J = J + 1
IF (J .LE. NEV) GOTO 81
RETURN
END
subroutine ope (n,x,y)
implicit real (a-h,o-z)
dimension x(l),y(l)

* preforms y = a x for the example of the triangular grid
* matrix is (m*(m+1))/2
* icount: used to count the number of operations y=ax.

c m = number of nodes in each direction. dimension n of
m = (m+1)/2

cicount = 0.5d0/float(m-1)
ix = 0

do 20 i=1,m
   jmax = m-i+1
   ix = i+1
   t = 0.0d0

   c to get y=ax, look at contributions to node (i,j) component ix
   c from its neighbour nodes.

   c contributions from north and east
   c north: node (i,j+1) --
   if (j.eq.1) go to 2
   pd = cicount*(i+1)
   t = t+pd*x(ix+i)
   c if node is in first column double probability of particle coming
   c from north : if (i.eq.1) t = t+pd*x(ix+i)

   c contributions from south and west
   c south (node (i-1,j) --
   if (j.eq.1) go to 3
   pu = cicount*(i+j)
   t = t+pu*x(ix-i)

   c west (node (i-l,j) --
   if (i.eq.1) go to 20
   y(i) = t
   t = t+pu*x(ix-j+jmax-1)
20 continue
2001 continue
   o=1
   th=0.3
   y(1)=o*x(2)+th*x(3)
   y(2)=o*x(1)+th*x(4)
   y(3)=th*x(1)+o*x(4)
   y(4)=th*x(2)+o*x(3)
   icount = icount + 1
   return
end

*--------------------------------------------------------
* SUBROUTINE OPE
*--------------------------------------------------------